INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES RENNES

Computer Science Department

An Introduction to Logic Programming

Mireille Ducassé

Last revision March 2024

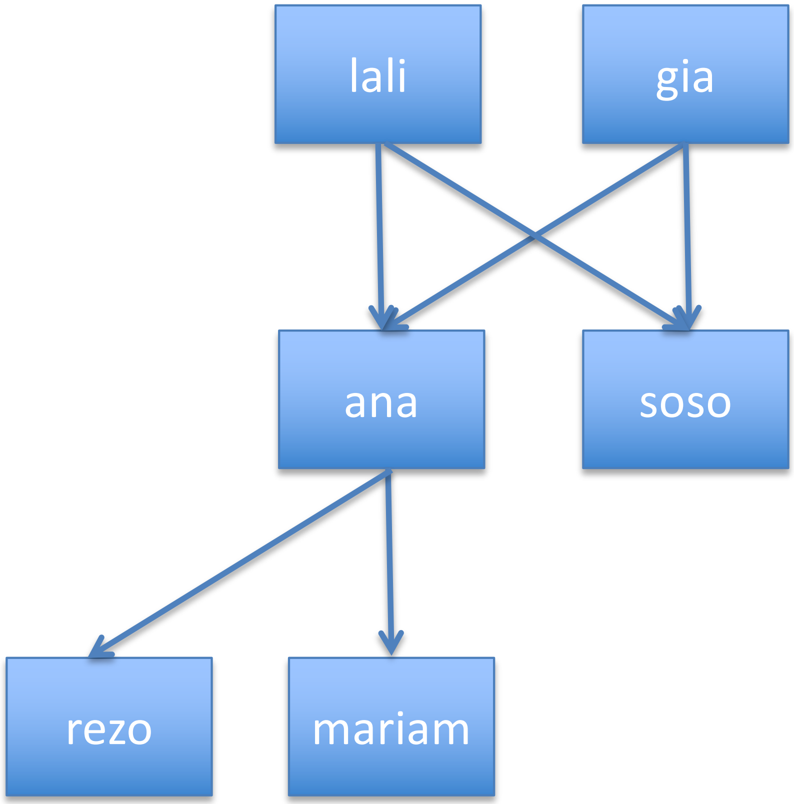
Is this a program ?

Draw the graph

```
lali   parent_of  soso.
lali   parent_of  ana.
gia    parent_of  ana.
gia    parent_of  soso.
ana    parent_of  mariam.
ana    parent_of  rezo.
```

Is this a program ?

lali parent_of soso.
lali parent_of ana.
gia parent_of ana.
gia parent_of soso.
ana parent_of mariam.
ana parent_of rezo.



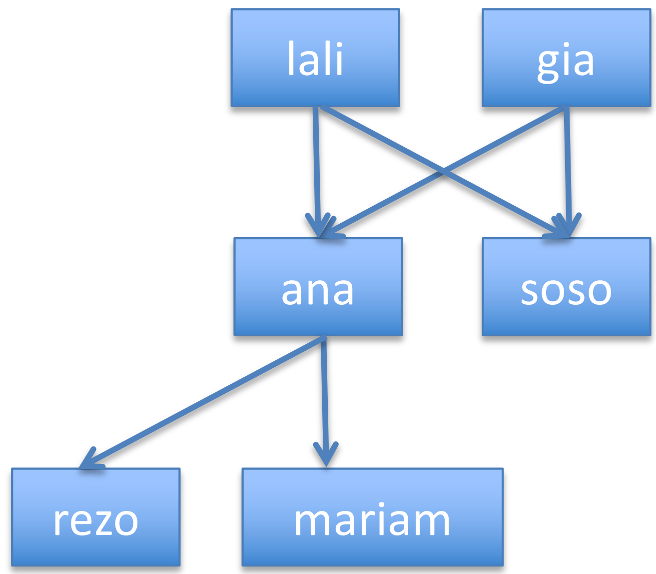
It is inde… g program !

Is lali a parent of ana ?

```
?- lali parent_of ana.
Yes

?- lali parent_of da…
No

?- lali parent_of X.
X = soso;
X = ana

?- Y parent_of ana.
Y = lali;
Y = gia
```

Of whom is lali a parent ?

```
lali   parent_of  soso.
lali   parent_of  ana.
gia    parent_of  ana.
gia    parent_of  soso.
ana    parent_of  mariam.
ana    parent_of  rezo.
```



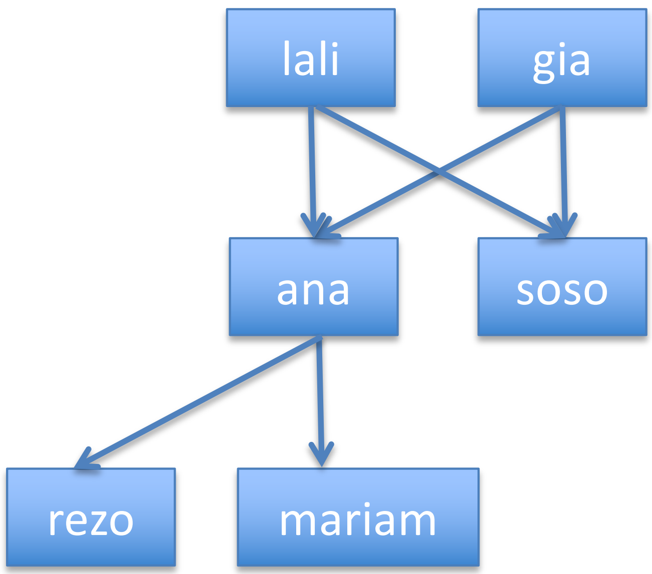
Enlarging the program

```
ancestor_of(A, C) :-
      parent_of(A, C).
ancestor_of(A, C) :-
      parent_of(A, X),
      ancestor_of(X, C).
```

A person A is an ancestor of another person C if either A is a parent of C or A is a parent of a third person X who is an ancestor of C

```
gia   parent_of  soso.
ana   parent_of  mariam.
ana   parent_of  rezo.
```

```
?- ancestor_of (lali, ana).
Yes
?- ancestor_of (lali, mariam).
Yes
?- ancestor_of (soso, mariam).
No
?- ancestor_of(lali, X).
X = soso ;
X = ana ;
X = mariam ;
X = rezo
```



Back to the program

6 Facts

```
lali   parent_of  soso.
lali   parent_of  ana.
gia    parent_of  ana.
gia    parent_of  soso.
ana    parent_of  mariam.
ana    parent_of  rezo.

ancestor_of(A, C) :-
     parent_of(A, C).
ancestor_of(A, C) :-
     parent_of(A, X),
     ancestor_of(X, C).
```

2 Rules

- 8 clauses
  - 6 facts, 2 rules
  - Terminated by a « . »
- 2 **predicates**
  - parent_of
  - ancestor_of
- 6 atoms
  - lali, ana, soso, gia, mariam, rezo
  - They are constants
- 5 Variables
  - A (twice), C (twice), X
  - **Begin with uppercase**
  - **Local to a clause**

Back to the program

```
lali   parent_of  soso.
lali   parent_of  ana.
gia    parent_of  ana.
gia    parent_of  soso.
ana    parent_of  mariam.
ana    parent_of  rezo.

ancestor_of(A, C) :-
    parent_of(A, C).
ancestor_of(A, C) :-
    parent_of(A, X),
    ancestor_of(X, C).
```

Implication (⇐)
Head is true if body if true

Conjunction (and)

Head

Body

Exercise ancestor 1/5

- Write the “ancestor_of” Prolog code corresponding to **your own family**,
  - going back to great grand parents
  - Include at least sisters, brothers, cousins, aunts and uncles
  - Start from the “ancestor.pl” file in the Moodle page
- Run the program
  - on EclipseClp
  - or SWI Prolog
    - Command : **`swipl`**
    - **`Or Online version : https://swish.swi-prolog.org`**

Exercise ancestor 2/5

- Test it

?- ancestor_of(<your grandma's name here>, <your name here>).

?- ancestor_of(A, <your name here>).

?- ancestor_of(<your name here>, C).

?- ancestor_of(A, C).

- Remember to keep the results of the test in the same file
  - in between comments

```
/*
< your tests here>
*/
```

Prolog terms

- Constants
  - Atoms
  - Numbers
- Strings
- Variables
- Lists
  - See next chapters
- Functors + arguments
  - Ex: parent_of(lali, X),
  - Ex: whatever(Name, another(Y), 3)
  - Number of arguments : **arity**
    - parent_of/2
    - whatever/3
- Queries and answers
  - Ex: ?- parent_of(lali, ana). YES

Exercise ancestor 3/5

- For your personal "ancestor_of" Prolog code corresponding to **your own family**, list
  - Constants
  - Strings
  - Variables
  - Lists
  - Functors with their arity
  - Some possible queries

Exercise ancestor 4/5

- Now program rules to define
  - **sibling/2**
    - a sibling of X is a child of X's parents but not X
  - **aunt_or_uncle/2**
    - an aunt or uncle of X is a sibling of a parent of X
  - **cousin/2**
    - first write the corresponding English sentence
    - "a cousin of X is …
  - **grandparent/2**
    - A grand parent of X is…
  - **greatgrandparent/2**
- **Test each predicate as soon as you have written it**
  - to **check** relations that are **correct**
  - to **check** relations that are **NOT correct**
  - to **find** relations
- **Keep the code and the tests in a file**
  - to be uploaded before next lecture on the Moodle page

Your programs

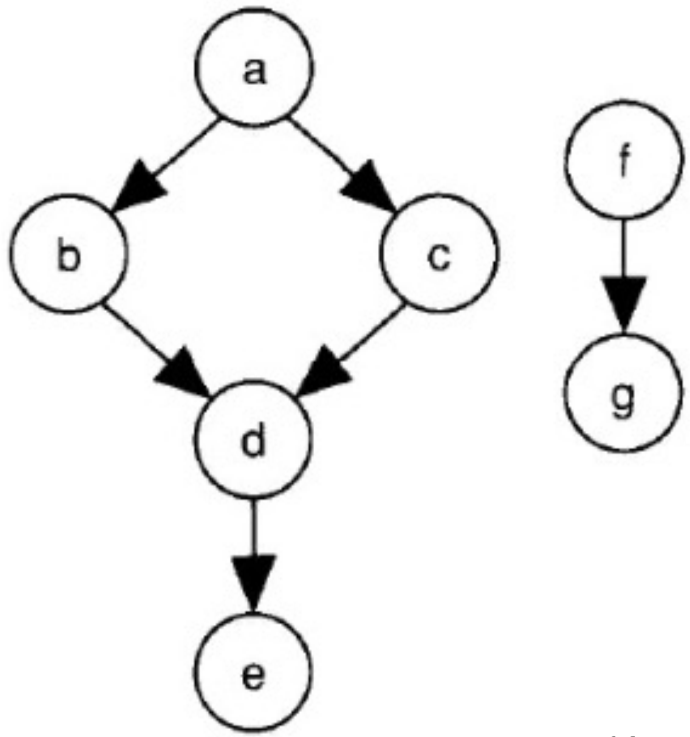
- Upload them on Moodle after each lecture
  - on the dedicated slots

- Make sure to add how you tested them

```
/*
 <program description>
*/

<Prolog code>

/*
?- <tested goals and results>
*/
```

Exercise 2.1

- Write two Prolog predicates that define connected nodes in the given graph:
  - `edge(Node1, Node2)`: it is true that there is a direct link between Node1 and Node2
  - `connected(Node1, Node2)` : there exists a path between Node1 and Node2
  - A node is considered connected to itself
- **Test each predicate**
  - to **check** connections that are **correct**
  - to **check** connections that are **NOT correct**
  - to **find** connections



Exercise 2.1 (bis)

```
edge(a, b).
edge(a, c).
edge(b, d).
edge(c, d).
edge(d, e).
edge(f, g).
```

```
connected(Node, Node).
connected(Node1, Node2) :-
    edge(Node1, Link),
    connected(Link, Node2).
```

Note that this is almost the same program as ancestor/2 with parent/2

Do you recognize this rule ?

```
toto(A, B) :-
    tutu(X, A),
    tutu(X, B),
    A \= B.
```

- Names of predicates and variables are crucial for us, human being, to be able to **read and understand** a program
- Prolog interpreter does not care as long as the **naming is consistent**

Do you recognize this rule ? (bis)

```
toto(A, B) :-
    tutu(X, A),
    tutu(X, B),
    A \= B.
```

If tutu is “parent_of”, this is the rule for sibbling/2

- Names of predicates and variables are crucial for us, human being, to be able to **read and understand** a program
- Prolog interpreter does not care as long as the **naming is consistent**

DECLARATIVE PROGRAMMING

Declarative programming

- We defined **what** is a parent and an ancestor
- We used the program with different « modes »
  - All parameters instantiated
    - Verification
  - Some parameters or none instantiated
    - Result generation
    - No predefined input or output

➢ Very powerful programming

```
lali   parent_of  soso.
lali   parent_of  ana.
gia    parent_of  ana.
gia    parent_of  soso.
ana    parent_of  mariam.
ana    parent_of  rezo.

ancestor_of(A, C) :-
       parent_of(A, C).
ancestor_of(A, C) :-
       parent_of(A, X),
       ancestor_of(X, C).
```

Basic mechanisms

The « magic » comes from

– **Unification**
  - ex: p(23, Y) = p(X, hello) with X/23 and Y/'hello'

and

⚠ In Prolog, "=" denotes unification NOT equality

– **Search tree and Backtracking**
  - search for (more/all) solutions upon failure

Unification (=)

1. If T_1 and T_2 are constants, then T_1 and T_2 unify if they are the same atom, or the same number
2. If T_1 is a variable and T_2 is any type of term, then T_1 and T_2 unify, and T_1 is instantiated to T_2 (and vice versa)
3. If T_1 and T_2 are complex terms then they unify if:
   1. They have the same functor and arity, and
   2. all their corresponding arguments unify, and
   3. the variable instantiations are compatible.

Unification examples

```
?- lali = lali.
Yes
?- lali = ana.
No
?- lali = X.       % Can 'lali' be unified with a free variable ?
X = lali
?- parent_of(lali, X) = ancestor_of(lali, ana).
No
?- parent_of(lali, X) = parent_of(lali, ana).
X = ana
?- parent_of(lali, X) = parent_of(Y, ana).
X = ana
Y = lali
```

More examples

```
?- X = lali, X = ana.
No
?- [X | Y] = [a, b, c]
X = a
Y = [b, c]
?- [a | Y] = [X , b, c].
X = a
Y = [b, c]
?- [a | Y] = [X , b | Z].
X = a
Y = [b | Z]
```

The unification algorithm of Robinson

- Input
  - 2 terms T1 and T2 to be unified
- Output
  - θ the most general unifier of T1 and T2
  - or failure
- Initialisation
  - θ : = ∅ , empty substitution
  - stack : = [T1 = T2]
  - failure : = false

Unification !

Unification algorithm 2/2

- while the stack is not empty and `not failure`, pop X = Y, case of
  - X is a variable not occurring in Y:
    substitute X by Y in the stack and in θ ;
    add X / Y in θ
  - Y is a variable not occurring in X:
    substitute Y by X in the stack and in θ ;
    add Y / X in θ
  - X and Y are constants or identical variables: go on
  - X=f(X1, .., Xn) and Y=f(Y1, .., Yn) for a functor f and n > 0 :
    push Xi=Yi, i=1..n
  - else `failure := true`
- end-while
- if `failure` then return failure else return θ

Exercise 2.2

- Use the previous algorithm to (try to) unify
  - Lali and soso
  - parent_of(lali, X) and Foo
  - 3 and 2+1
    - Hint : 2+1 is syntatic sugar for +(2, 1)
  - parent_of(lali, X) and parent_of(Y, ana, Z)
  - parent_of(lali, X) and parent_of(Y, ana)
  - f(A,A) and f([3, 2],C)
  - father(X) and X

Avoiding infinite terms : Occurs check

Without occurs check :

?- father(X) = X.

X=father(father(father(father(father(father
(father(father(father(father(father(father
(father(father(father(father(father(father
(father(father(father(father(father(father
(father(father(father(father(father(father
(father(father(father(father(father(father
(father(father(father(father(father(father
(father(father(father(father(father(father
(.............

ISO standard Prolog : **the two terms do not unify**

Some Prolog systems handle infinite terms

Exercise 2.3

- First “guess” the result then use the algorithm to try to unify
  - 6 and 2*3
  - edge(a, X) and edge(Y, b, Z)
  - edge(a, X) and edge(Y, b)
  - connected(a, X) and connected(Y, e)
  - foo(A,A) and foo(bar(B),C)
  - p(X) and X

Exercise ancestor 5/5

- So far we have identified people by their first name
  - It is not a unique identifier
  - In a real genealogy program we need more information about them
- Update your program and represent each person by a functor p(FirstName, YearOfBirth)
  - You must modify parent_of/2
  - Do you need to modify ancestor_of/2 ?
    - Why or why not ?
  - Give examples of query.
- Is this sufficient to identify uniquely a person ?
- Do not forget to test your solution

Exercise ancestor 5/5 (bis)

```
parent_of(p(lali,  1950),  p(soso, 1973)).
parent_of(p(lali,  1950),  p(ana, 1975)).
parent_of(p(gia,  1950),  p(ana, 1975)).
parent_of(p(gia,  1950),  p( soso, 1973)).
parent_of(p(ana, 1975),  p(mariam, 2000)).
parent_of(p(ana, 1975),  p(rezo, 2002)).

ancestor_of(A, C) :-
   parent_of(A, C).
ancestor_of(A, C) :-
   parent_of(A, X),
   ancestor_of(X, C).

/* examples of query
?- ancestor(A, X).
?- ancestor(p(lali,  1950), X).
?- ancestor(p(lali,  1950), p(Y, 1973)).
*/
```

Unification can deal with functors

For a reel program you would need more information and a unique identifier

For example

```
p(p1, lali,    1950, georgia).
p(p2, gia,     1950, georgia).
p(p3, soso, 1973, georgia).
p(p4, ana,    1975, france).
p(p5, mariam, 2000, france).
p(p6, rezo,   2002, france).

parent_of(p1, p3).
parent_of(p1, p4).
parent_of(p2,  p3).
parent_of(p2,  p4).
parent_of(p4, p5).
parent_of(p4, p6).
```

Identifiers here are already better than simple numbers

Or better

```
p(p1Lali1950,       lali,    1950, georgia).
p(p2Gia1950,        gia,     1950, georgia).
p(p3Soso1973,      soso,  1973, georgia).
p(p4Ana1975,        ana,     1975, france).
p(p5Mariam2000, mariam, 2000, france).
p(p6Rezo2002,      rezo,    2002, france).

parent_of(p1Lali1950,  p3Soso1973).
parent_of(p1Lali1950,  p4Ana1975).
parent_of(p2Gia1950,  p3Soso1973).
parent_of(p2Gia1950,  p4Ana1975).
parent_of(p4Ana1975, p5Mariam2000).
parent_of(p4Ana1975, p6Rezo2002).
```

Unique identifiers that also convey **readable** information **thanks to atoms**

Prolog search tree

Definition: A search tree of a goal ***G*** with respect to a program ***P*** :

- The root is ***G***
- Nodes are goals (*resolvent*), with one selected goal
- There is an edge from a node ***N*** for each clause in the program whose head **unifies with the selected goal** of ***N***
  - edges are labeled by the current substitution

Remarks

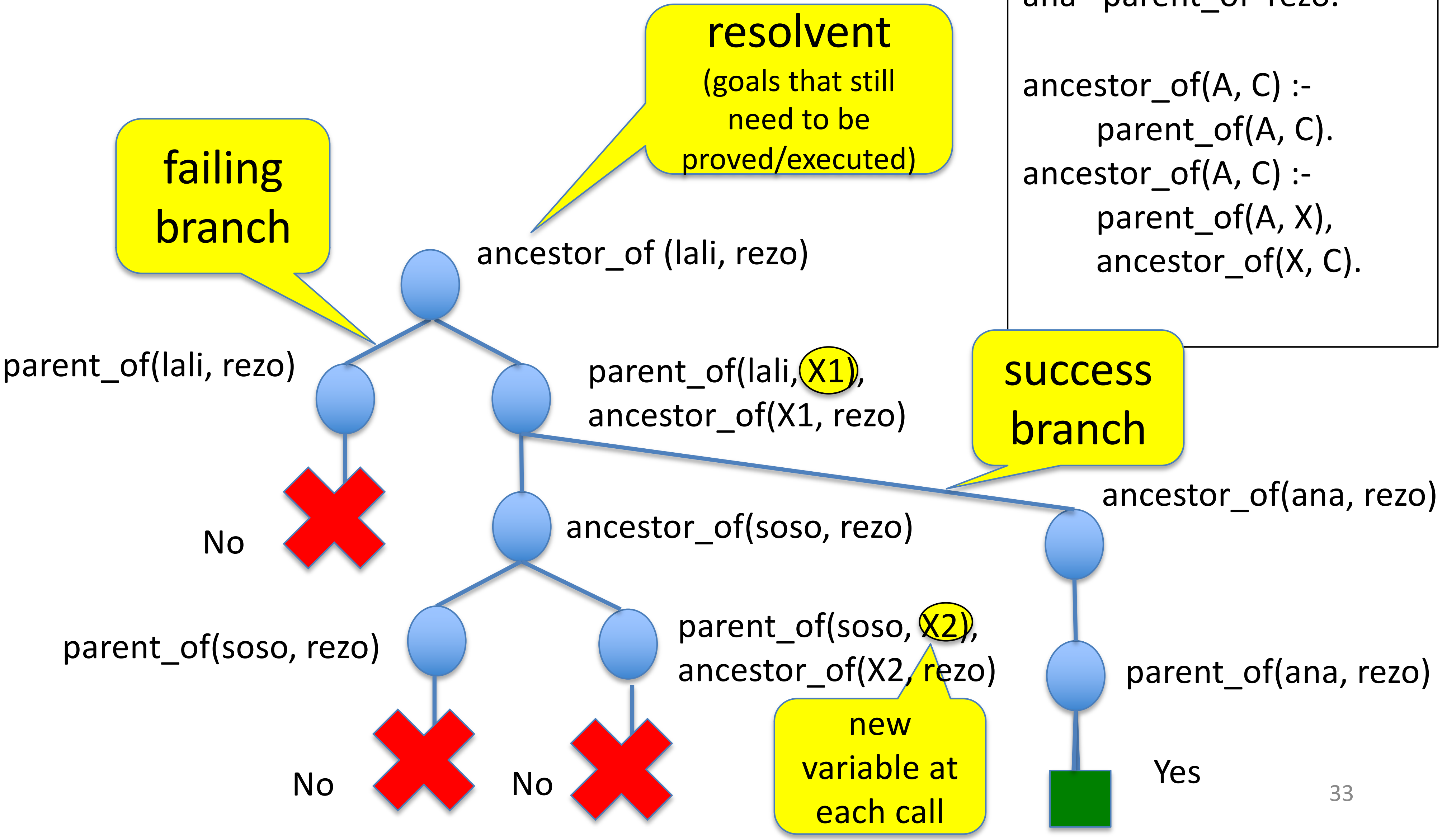
- Each branch from the root is a computation of ***P*** by ***G***
- Leaves are
  - success nodes, where the empty goal has been reached, or
  - failure nodes, where the selected goal cannot be further reduced
- Success nodes correspond to solutions of the root

Search tree

?- ancestor_of (lali, rezo).

Yes

```
lali   parent_of  soso.
lali   parent_of  ana.
gia    parent_of  ana.
gia    parent_of  soso.
ana    parent_of  mariam.
ana    parent_of  rezo.

ancestor_of(A, C) :-
       parent_of(A, C).
ancestor_of(A, C) :-
       parent_of(A, X),
       ancestor_of(X, C).
```



Exercise 2.4

- What happens if we exchange the first 2 lines of the program ?

```
?- ancestor_of (lali, rezo).

lali   parent_of  soso.
lali   parent_of  ana.
gia    parent_of  ana.
gia    parent_of  soso.
ana    parent_of  mariam.
ana    parent_of  rezo.

ancestor_of(A, C) :-
        parent_of(A, C).
ancestor_of(A, C) :-
        parent_of(A, X),
        ancestor_of(X, C).
```

```
?- ancestor_of (lali, rezo).

lali   parent_of  ana.
lali   parent_of  soso.
gia    parent_of  ana.
gia    parent_of  soso.
ana    parent_of  mariam.
ana    parent_of  rezo.

ancestor_of(A, C) :-
        parent_of(A, C).
ancestor_of(A, C) :-
        parent_of(A, X),
        ancestor_of(X, C).
```

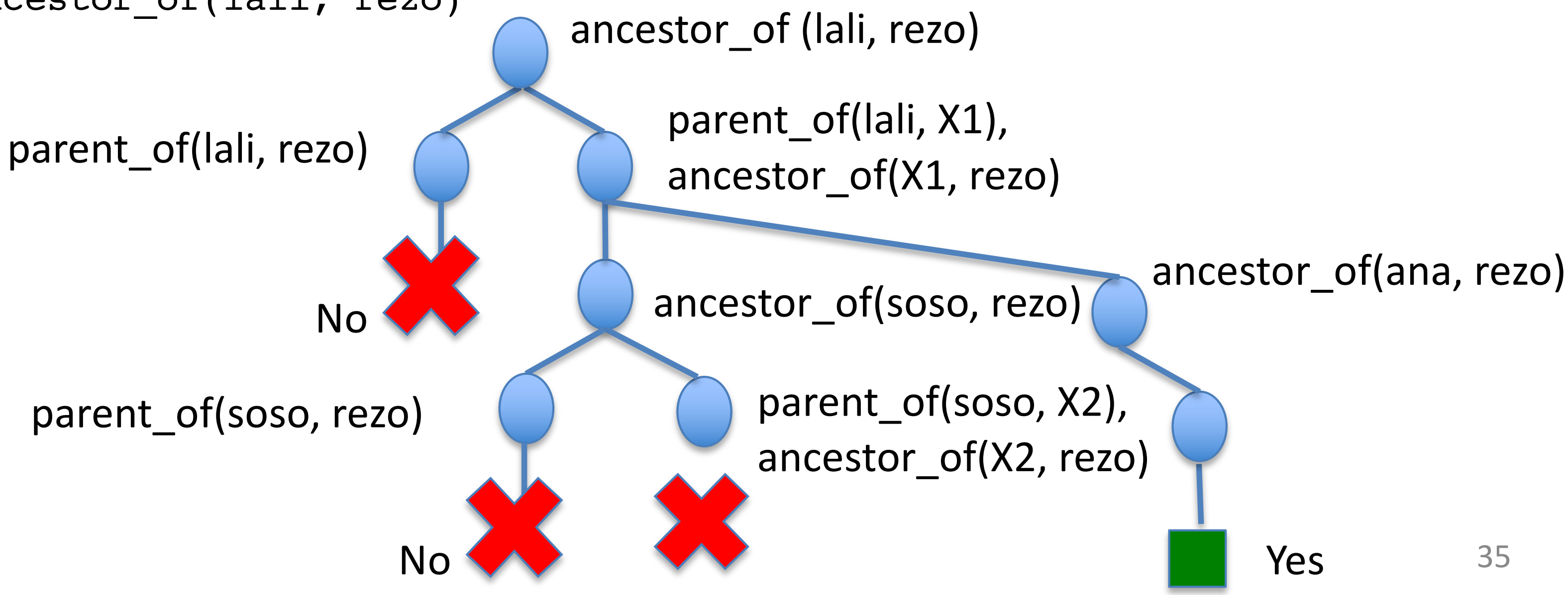
Does it change the result ?

Does it change the search tree ?

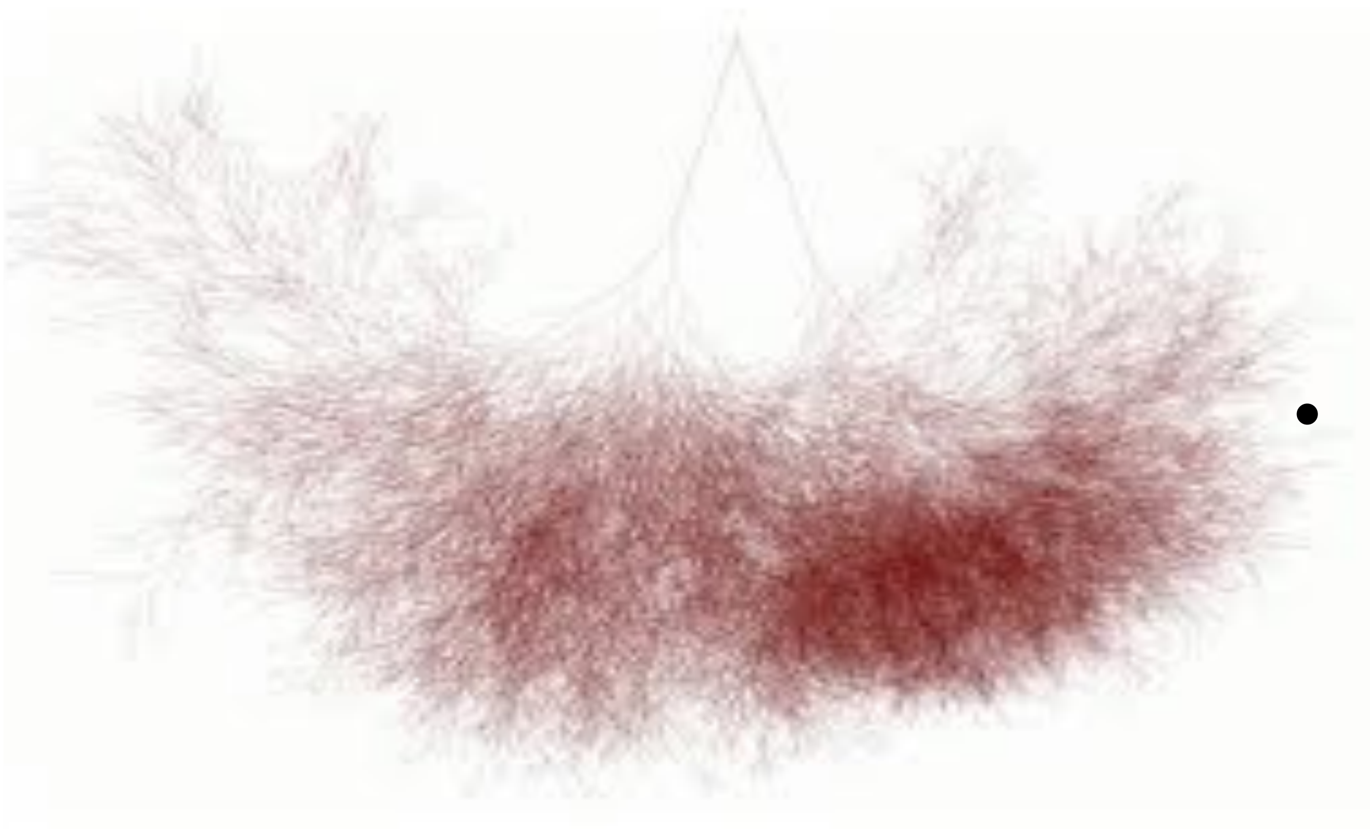
Prolog trace

```
ancestor_of(lali, rezo).
  (1) 1 CALL  ancestor_of(lali, rezo)
  (2)  2 CALL  lali parent_of rezo
  (1) 1 NEXT  ancestor_of(lali, rezo)
  (3)  2 CALL  lali parent_of _298
  (3)  2 *EXIT  lali parent_of soso
  (4)  2 CALL  ancestor_of(soso, rezo)
  (5)   3 CALL  soso parent_of rezo
  (5)   3 FAIL  ... parent_of ...
  (4)  2 NEXT  ancestor_of(soso, rezo)
  (6)   3 CALL  soso parent_of _535
  (6)   3 FAIL  ... parent_of ...
  (4)  2 FAIL  ancestor_of(..., ...)
  (3)  2 REDO  lali parent_of _298
  (3)  2 EXIT  lali parent_of ana
  (7)  2 CALL  ancestor_of(ana, rezo)
  (8)   3 CALL  ana parent_of rezo
  (8)   3 EXIT  ana parent_of rezo
  (7)  2 *EXIT  ancestor_of(ana, rezo)
  (1) 1 *EXIT  ancestor_of(lali, rezo)
```

```
lali  parent_of  soso.
lali  parent_of  ana.
gia   parent_of  ana.
gia   parent_of  soso.
ana   parent_of  mariam.
ana   parent_of  rezo.

ancestor_of(A, C) :-
        parent_of(A, C).
ancestor_of(A, C) :-
        parent_of(A, X),
        ancestor_of(X, C).
```



Execution trees can be large



- Prolog run time system can cope for **hundreds of thousands** of nodes
- When search space too large
  - time to consider **Constraint Logic Programming**

MORE ON PROLOG BASICS

Ex 2.6: Which queries are satisfied ?

```
house_elf(dobby).
witch(hermione).
witch('McGonagall').
witch(rita_skeeter).
magic(X):-
    house_elf(X).
magic(X):-
    wizard(X).
magic(X):-
    witch(X).
```

1. ?- magic(house_elf).
2. ?- wizard(harry).
3. ?- magic(wizard).
4. ?- magic('McGonagall').
5. ?- magic(Hermione).

Another example : successor

Suppose we use the following way to write numerals:

1. **0** is a numeral.
2. If **X** is a numeral, then so is **succ(X)**.

```
numeral(0).
numeral(succ(X)):-
    numeral(X).
```

```
/*

?- numeral(succ(succ(succ(0)))).
yes

?- numeral(succ(1)).
No

?- numeral(2).
No

?- numeral(X).
X=0;
X=succ(0);
X=succ(succ(0));
X=succ(succ(succ(0)));
X=succ(succ(succ(succ(0))))

*/
```

Exercise 2.7 : addition 1/2

Write a program that adds two numbers represented with functor succ/1

```
?- add(succ(succ(0)),succ(succ(succ(0))), Result).
Result = succ(succ(succ(succ(succ(0)))))
yes
```

Exercise 2.6 : addition 1/2 (bis)

Write a program that adds two numbers represented with functor succ/1

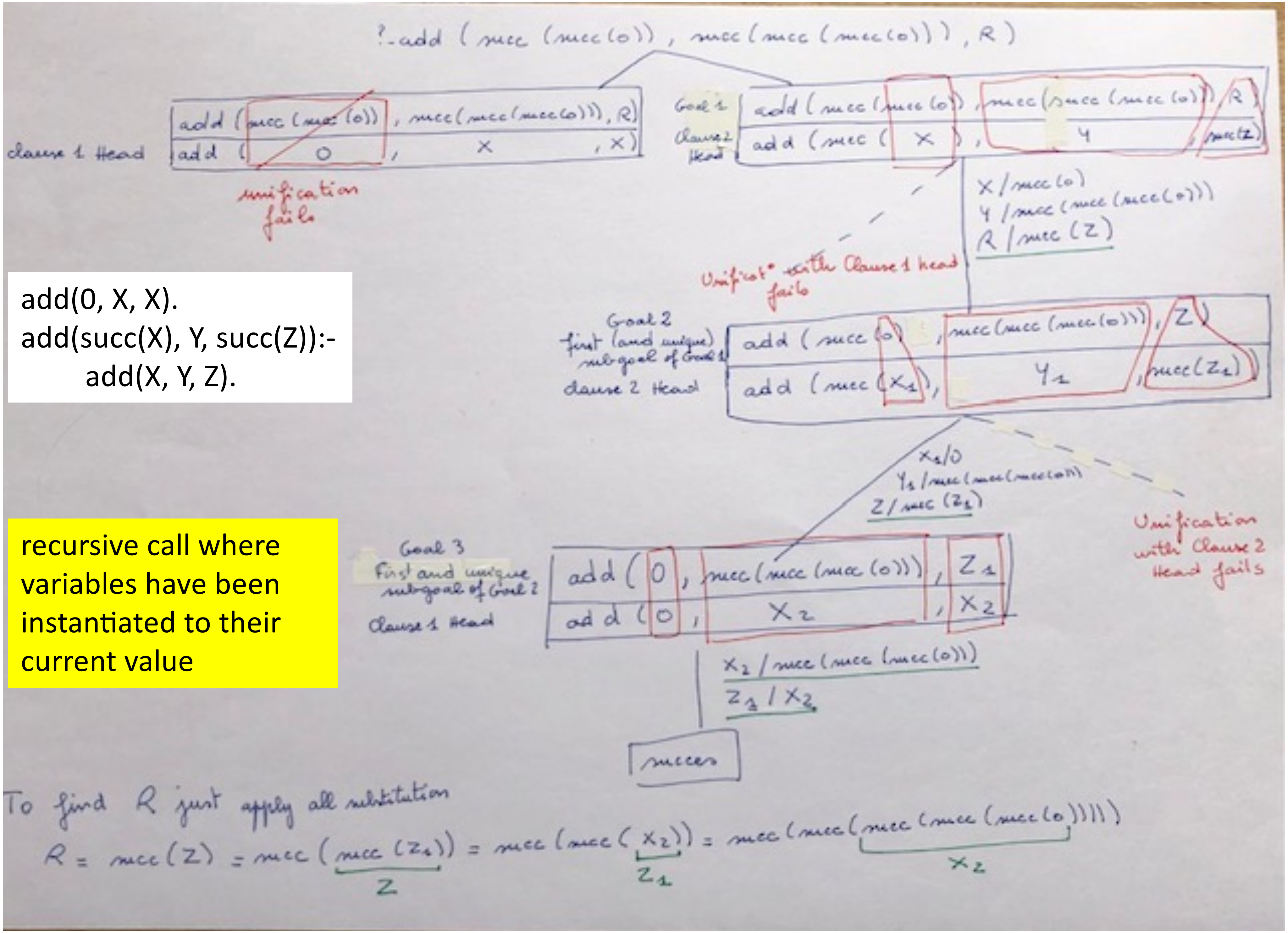
```
?- add(succ(succ(0)),succ(succ(succ(0))), Result).
Result=succ(succ(succ(succ(succ(0)))))
yes
```

```
add(0, X, X).
add(succ(X), Y, succ(Z)):-
    add(X, Y, Z).
```

Exercise 2.6 : addition 2/2

Build the **search tree** for the resolution of

?- add(succ(succ(0)),succ(succ(succ(0))), Result).




```
add(0, X, X).
add(succ(X), Y, succ(Z)):-
    add(X, Y, Z).
```

recursive call where variables have been instantiated to their current value

Exercise ancestor to be done at home and **uploaded on Moodle**

- Relative to the ancestor program given in the slides, give the answer of query

  ?- ancestor_of (lali, davit).

- Build the search tree for its resolution following the model of the previous search tree

Remember

- “Yes” and “No” are valid answers
  - this is **logic** programming

Unification exercise

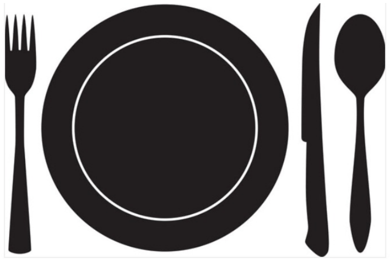
- (Try to) unify

  my_pred(foo(X), Y , bar(67, toto(A)) and

  my_pred(Foo , hello, bar(N , toto(72))

- Give the substitutions and the most general unifier

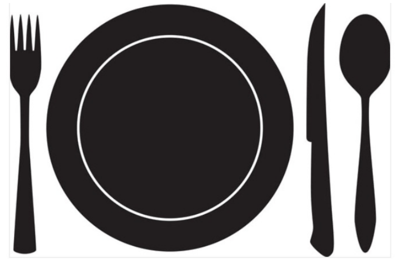
A French menu



- Typically:
  - Appetizer
  - Main course (meat, fish or vegetarian)
  - Dessert
- Examples of appetizers
  - salad, poached egg, artichoke
- Examples of main courses
  - meat: steak with vegetables, chicken with fries,
  - fish: trout with rice, salmon with eggplant
  - vegetarian: falafel with rice, vegetable lasagna
- Examples of desserts
  - fruit salad, fresh fruit, cake

French menu is the basis of the assessment project

Exercise French menu



Write a Prolog program

1. Facts to introduce components
2. Rule(s) to define/verify the structure of a French menu

A valid menu (test case -> YES)

?- french_menu(salad, trout_with_rice, cake).

Invalid menus (test cases -> NO)

?- french_menu(salad, trout, cake).

?- french_menu(falafel_with_rice, trout_with_rice, cake).

Take your time to search, code and test your own program

Then take your time to understand the following solution

Exercise French menu (bis)

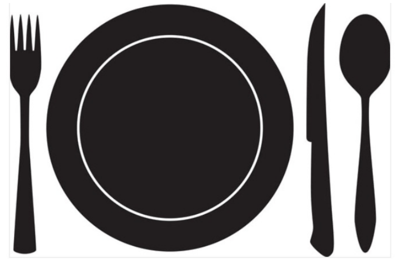
Facts

```
appetizer(salad).
appetizer(poached_egg).
appetizer(artichoke).

meat_course(steak_with_vegetables).
meat_course(chicken_with_fries).

fish_course(trout_with_rice).
fish_course(salmon_with_eggplant).

veggy_course(falafel_with_rice).
veggy_course(vegetable_lasagna).

dessert(fruit_salad).
dessert(fresh_fruit).
dessert(cake).
```

Rules

```
french_menu(A, M, D) :-
        appetizer(A),
        main_course(M),
        dessert(D).

main_course(M) :-
        meat_course(M).
main_course(M) :-
        fish_course(M).
main_course(M) :-
        veggy_course(M).
```

Exercise French Menu: Update 1



- Nowadays, people tend to eat less
- Restaurants often offer the possibility to take
  - Appetizer + main course, or
  - Main course + dessert, or
  - Appetizer + main course + dessert
- Update your program to take this into account

- Valid menus (test cases -> YES)

```
?- french_menu(salad, trout_with_rice).
?- french_menu(trout_with_rice, cake).
?- french_menu(salad, trout_with_rice, cake).
```

- Invent test cases -> NO

Ex French Menu : Update 1 (bis)

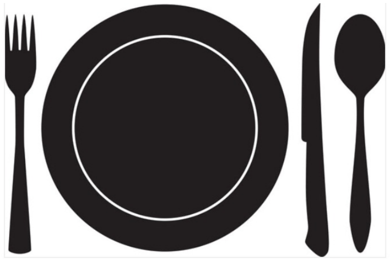
Facts

```
appetizer(salad).
appetizer(poached_egg).
appetizer(artichoke).

meat_course(steak_with_vegetables).
meat_course(chicken_with_fries).

fish_course(trout_with_rice).
fish_course(salmon_with_eggplant).

veggy_course(falafel_with_rice).
veggy_course(vegetable_lasagna).

dessert(fruit_salad).
dessert(fresh_fruit).
dessert(cake).
```

Rules

```
french_menu(A, M) :-
        appetizer(A),
        main_course(M).
french_menu(M, D) :-
        main_course(M),
        dessert(D).

french_menu(A, M, D) :-
        appetizer(A),
        main_course(M),
        dessert(D).

main_course(M) :-
        meat_course(M).
main_course(M) :-
        fish_course(M).
main_course(M) :-
        veggy_course(M).
```

Exercise French Menu : Update 2



- Sometimes cheese can replace dessert, sometimes it is offered before dessert
- Update your program to take this into account

- Valid menus (test case -> YES)

?- french_menu(salad, trout_with_rice).

?- french_menu(trout_with_rice, roquefort).

?- french_menu(salad, trout_with_rice, roquefort).

?- french_menu(salad, trout_with_rice, roquefort, cake).

- Invent test cases -> NO

Ex French Menu : Update 2 (bis)

Facts

```
appetizer(salad).
appetizer(poached_egg).
appetizer(artichoke).

meat_course(steak_with_vegetables).
meat_course(chicken_with_fries).

fish_course(trout_with_rice).
fish_course(salmon_with_eggplant).

veggy_course(falafel_with_rice).
veggy_course(vegetable_lasagna).

dessert(fruit_salad).
dessert(fresh_fruit).
dessert(cake).

cheese(roquefort).
cheese(camembert).
```

Rules

```
% french_menu/2
french_menu(A, M) :-
        appetizer(A),
        main_course(M).
french_menu(M, D) :-
        main_course(M),
        dessert_or_cheese(D).

% french_menu/3
french_menu(A, M, D) :-
        appetizer(A),
        main_course(M),
        dessert_or_cheese(D).

% french_menu/4
french_menu(A, M, C, D) :-
        appetizer(A),
        main_course(M),
        cheese(C),
        dessert(D).

main_course(M) :-
        meat_course(M).
main_course(M) :-
        fish_course(M).
main_course(M) :-
        veggy_course(M).

dessert _or_cheese(D) :-
        cheese(D).
dessert _or_cheese(D) :-
        dessert(D).
```

Hindsight

- Note how easy it is to increase the power of Prolog programs
    - **Powerful prototyping language**

Hanoi Towers program 1/2

```
move(1, Source, Target, _):-
    printf("Move top disk from %w  to %w\n", [Source, Target]).
move(N, Source, Target, Aux):-
    N > 1,                             % the 2 clauses are exclusive
    printf("Solve problem %w\n", [N]),
    N1 is N-1,
    move(N1, Source, Aux, Target),     % first solve N-1 problem
    move(1, Source, Target, _),        % to be able to move large ring
    move(N1, Aux, Target, Source).     % then really solve N-1 problem
```

Note the 2 recursive calls. The problem **IS** difficult.
This program explains it !
Note also predicate “printf/2” that can be very useful !

Hanoi Towers program 2/2

- Run the program with query

  ?- move(4, source, target, auxiliary).

- To understand the execution, play the game in parallel with https://www.mathplayground.com/logic_tower_of_hanoi.html

Hanoi Towers program 2/2 (bis)

```
move(1, Source, Target, _):-
      printf("Move top disk from %w  to
                %w\n", [A, B]).
move(N, Source, Target, Aux):-
      N > 1,
      printf("Solve problem %w\n", [N]),
      N1 is N-1,
      move(N1, Source, Aux, Target),
      move(1, Source, Target, _),
      move(N1, Aux, Target, Source).
```

```
?- move(4, source, target, auxiliary).
Solve problem 4
Solve problem 3
Solve problem 2
Move top disk from source  to auxiliary
Move top disk from source  to target
Move top disk from auxiliary  to target
Move top disk from source  to auxiliary
Solve problem 2
Move top disk from target  to source
Move top disk from target  to auxiliary
Move top disk from source  to auxiliary
Move top disk from source  to target
Solve problem 3
Solve problem 2
Move top disk from auxiliary  to target
Move top disk from auxiliary  to source
Move top disk from target  to source
Move top disk from auxiliary  to target
Solve problem 2
Move top disk from source  to auxiliary
Move top disk from source  to target
Move top disk from auxiliary  to target
```